International Journal of Security and Its Applications
Vol. 10, No. 9 (2016), pp.193-210
http://dx.doi.org/10.14257/ijsia.2016.10.9.20

A Provably-Correct Micro-Dalvik Bytecode Verifier

Jiang Nan[1,3*], He Yanxiang[1,2], Zhang Xiaotong[1], Liu Rui[1] and Shen Yunfei[1]

[1]*Computer School, Wuhan University, China*
[2] *State Key Laboratory of Software Engineering (Wuhan University), China*
[3] *Computer School, Hubei University of Technology, China*
[1]*nanjiang@whu.edu.cn*, [2]*yxhe@whu.edu.cn*

Abstract

In this paper we propose and implement a provably-correct bytecode verifier for Micro-Dalvik which is a significant subset of Dalvik. We take the approach of a data flow analysis on semilattice to solve the bytecode verification. The operational semantics of Micro-Dalvik is developed. The semilattice structure is constructed and transfer functions are defined. Based on the semilattice and transfer functions the well-typing is described. The bytecode verifier is realized to compute a method type to see whether each instruction is assigned a well-typing. By giving conforming relations between the machine state and the well-typing, the correctness of the verifier is proved that the verifier guarantees safe execution and well-typed Micro-Dalvik programs do not produce type errors.

Keywords: *bytecode verifier, Dalvik, type system*

1. Introduction

Applications on Android mobile device platform are written in Java and compiled to the Dalvik bytecode. Since applications might be from a malicious party or be corrupted during network transmission, bytecode verification is essential. Other reasons for needing a bytecode verifier include that the Dalvik optimizer can work on a safer mode with verified bytecode and Dalvik can perform an exact garbage collection since improper register use has already been caught by the verifier.

The Dalvik bytecode verifier attempts to provide the same set of checks and guarantees as are described in the Java Virtual Machine Specification (JVM) [1]. However, different from the JVM which is stack-based, Dalvik is register-based and does not have an operand stack. It approximately imitates common real machine architectures and C-style calling conventions [2]. Therefore, the implementation of the Dalvik bytecode verifier is very different from that of the JVM bytecode verifier. Checks relevant to the operand stacks are not performed. On the other hand, there are some new ones for the Dalvik bytecode verifier.

There is a large body of literature on formalization of JVM bytecode verification [3-9], among which a noticeable work is done by Klein and Nipkow who introduce a unified model of a Java-like source language, VM, and compiler and demonstrate a state-of-the-art machine-checked language definitions. However, there is a relatively small amount of formalization of the VM with a register-based architecture, e.g. Dalvik. Payet et al. define an operational semantics of the Dalvik bytecode and model Android platform in order to develop a symbolic executor for Android applications [10]. In their formalization the Dalvik bytecode is assumed to be verified. Jeon et al. also give and evaluate a symbolic executor SymDroid [11], but they do not consider bytecode verification either.

*Corresponding Author

ISSN: 1738-9976 IJSIA

In this paper we propose and implement a provably-correct bytecode verifier for Micro-Dalvik in the theorem proof assistant Isabelle/HOL using the unified model described by Klein and Nipkow. The Micro-Dalvik is a significant subset of Dalvik. To model such a register-based architecture and obtain a provably-correct bytecode verifier, our distinct work is listed as follows.

- The operational semantics of Micro-Dalvik. The absence of the operand stack makes Dalvik pass two special values, *i.e.*, the return value and the raised exception object, through the interpreter itself instead of pushing them onto the top of the operand stack, which results in a different formalization of the state space. Together with a different instruction set and method calling convention the operational semantics of Micro-Dalvik is completely different from that of any stack-based VM.

- A type system for Micro-Dalvik. The different formalization for the state space brings about different formal description of its type information. This type system includes the semilattice structure of the state type and transfer functions in terms of types. The well-typings based on ordering of the semilattice and transfer functions are defined.

- An executable Micro-Dalvik bytecode verifier and Type-safety proof for the Micro-Dalvik type system. This verifier is a concrete version of a general framework of the Kildalls algorithm used for type inference. The type-safety theorem states that well-typed Micro-Dalvik programs do not produce type errors. A defensive Micro-Dalvik is defined which performs type and sanity checks. The conformance relations between the the Micro-Dalvik machine state and program type are specified. By proving type-safety the correctness of the bytecode verifier is proved.

We describe the above four issues in Section 2 through 4, respectively. Finally we discuss related work in Section 5 and conclude in Section 6.

2. The Operational Semantics of Micro-Dalvik

2.1. The Micro-Dalvik Instructions and Programs

Aiming at performing formal analysis and obtaining a clean semantics, we generalize the Dalvik instruction set into 20 instructions. The instruction set (of HOL type *instr*) of Micro-Dalvik is as below (we omit threads and arrays at present):

instr = **Const** *nat nat*
| **Iget** *nat nat vname vname*
| **Iput** *nat nat vname vname*
| **AriBinop** *ariop nat nat nat*
| **Cmpeq** | **Cmp**ne | **Cmp**gt | **Cmp**ge | **Cmp**lt | **Cmp**le *nat nat nat*
| **IFfalse** *idx int*
| **Goto** *int*
| **Return** *nat*
| **Invoke** *nat nat mname*
| **NewIntance** *nat cname*
| **Throw** *nat*
| **CheckCast** *nat cname*
| **Move** *nat nat*
| **MoveResult** *nat*
| **MoveException** *nat* (1)

These abstract instructions have a direct correspondence to the Dalvik instructions. In contrast, the Micro-Dalvik involves two kinds of generalization.

- Operations differing only in operand types are generalized into one instruction.

- Instructions differing only in the number of bits to represent register indices are generalized into one instruction.

We employ the unified model described by Klein and Nipkow[3] to model Micro-Dalvik programs. Some type synonyms are introduced. The three types, *cname*, *vname*, and *mname* are all for type *string*. The types *addr* and *pc* are both for type *nat*. The datatype *val* can be *Unit*, *Null*, *Bool bool*, *Intg int*, and *Addr nat*. A Micro-Dalvik value is of type *val*. The datatype *ty* can be *Void*, *Boolean*, *Integer*, *NT*, and *Class cname*. A Micro-Dalvik type is of type *ty*. Type variables are written *'a*, *'b*, *etc*. The datatype *'a option=None | Some a* adjoins a new element *None* to a type *a*, Some *x* written as ⌊*x*⌋. A partial function (called a map) *f x*=⌊*y*⌋ means that *x* is mapped to *y*. The partial function update *f*(*x*:=⌊*y*⌋) is written *f*(*x*↦*y*). Three infix constructors of lists (type *'a list*) are *x*#*xs*, *xs*!*i* and *xs*@*ys*. The first puts an element *x* in front of a list *xs*; the middle is *i th*-element of *xs*; the last appends two lists, *xs* and *ys*. A empty list is written []. The operation *hd xs* is the first element of *xs*, and the *tl xs* is the rest of *xs*.

Bytecode verification is concerned with the abstract interpretation of bytecode methods, and hence we will explain the formalization of the Micro-Dalvik method body in detail.

The type synonym *dvm_mb = nat* × *instr list* × *ex_table* defines the Micro-Dalvik method body as a tuple, (reg_0,*ins*,*xt*) consisting of one number, a list of instructions and an exception table. The reg_0 is the number of registers (not counting arguments at the last *N* registers) which is known at compile time. We do not consider optimization at present. The first *N* registers are used to store this pointer and parameters by Move instructions generated by the compiler, which are immediately followed by local variables defined in this method in order. The rest of registers works as a space like the operand stack of a stack-based VM. The exception table *xt*:: (*pc* × *pc* × *cname* × *pc*) *list*, is a list of quadruples (*s*,*e*,*xc*,*h*) denoting an exception table entry. The s and the result of e minus 1, are program counters pointing to the first and the last instructions of the corresponding try block in the source level, respectively. The number *h* points to the first instruction, i.e., *MoveException*, in an exception handler. The class name *xc* represents a class of exceptions that this exception handler is designated to catch, meaning that the exception handler will work only if the thrown exception is an instance of the given exception class or one of its subclasses. The direct subclass relation is inductively defined as a set of pairs. The subclass relation is its reflexive transitive closure supported by Theory Transitive_Closure of HOL.

2.2. The Operational Semantics of Micro-Dalvik

2.2.1. Machine State: Each frame of the Dalvik VM consists of a particular number of registers as well as some data needed to execute the method. It does not have an operand stack which is used as a work space, preparing arguments to be passed to methods, and especially, receiving the method result value and the exception object raised. The absence of the operand stack in the Dalvik VM complicates the machine state. We formalize its machine state as a 4-tuple (*xp*,*h*, *frs*, ret_v):

$$dvm\_state = addr\ option \times heap \times frame\ list \times val, \tag{2}$$

where *xp* is a flag indicating whether an exception object is thrown: a reference to the exception object if Yes, or None; *h* is a map from memory locations to objects; ret_v represents the return value; *frs* is a list of frames. A frame is defined as a 6-tuple (xc_v,*regs*,*C*,*M*, *ts*,*pc*) of type:

$$frame = val \times val\ list \times cname \times mname \times ty\ list \times pc, \tag{3}$$

where xc_v is of the form *Addr n* in which n represents a reference to the just-caught exception object (*n* is of type *nat*). The list *regs* represents values in registers. The last *N* registers store the *N* arguments to the called method specified in the *Invoke* instruction of the calling method. The number *pc* points to the current instruction. We also store the class name *C*, the method name *M* and parameter types *ts* which we use to know which method each frame belongs to.

The *seesmethods* is an inductively defined predicate which transverses the class hierarchy upwards starting from *C* and ending with the class *Object*. The inductive case is the rule:

$$\left.\begin{array}{l} map\_of\ P\ C = \lfloor (D, fs, ms) \rfloor; C \neq Object; seesMethods\ P\ D\ Mmts; \\ Mmts' = Mmts\ +\!+\ (map\_option\ (\lambda m.\ (m, C)) \\ \circ (map\_of\ (map\ (\lambda (x, y, z).\ ((x, y), z))\ ms))) \\ seesMethods\ P\ C\ Mmts'. \end{array}\right\} \Rightarrow \quad (4)$$

The result of *seesmethods P C* represents a set of maps $(M,ts) \mapsto ((t,mb),C')$. Each map means that from *C*, a method *M* with parameter types, *ts*, defined in *C'*, is visible to *C*. This definition takes overriding into account via a ++ operation. The overriding requires that both the name and the parameter types of two methods are the same by turning a 4-tuples (*M*,*ts*,*t* ,*mb*) into a 3-tuple ((*M*,*ts*),*t*,*mb*)) via a *lambda* calculation. Using this inductively defined predicate, we define two functions, *seesMethod* and *method* as below which will be used for the rest:

$$\begin{array}{l} seesMethod\ P\ C\ M\ ts\ t\ m\ D \equiv \\ \quad \exists Mmts.\ seesMethods\ P\ C\ Mmts \wedge Mmts\ (M, ts) = \lfloor ((t, m), D) \rfloor \\ method\ P\ C\ M\ ts \equiv THE(D, t, mb).\ seesMethod\ P\ C\ M\ ts\ t\ m\ D \end{array} \quad (5)$$

2.2.2. The Operational Semantics: The core function *exec_instr* defines the single step execution. It takes the instruction to execute, the program *P*, the heap *h*, the just-caught exception object reference *x*, local variables *reg* of the current call frame, the class *C*, the method name, *M* and parameter types *ts* of the method that is currently executed, the current *pc*, the rest of the call frame stack *frs*, and the return value *r* as parameters. It specifies that in a program *P*, starting the state (*None*,*h*, (*x*,*reg*,*C*,*M*,*ts*,*pc*)#*frs*,*retv*), executing an instruction updates the state to a new state. Let us look at the most complicated rule for the method invocation instruction:

$$\begin{array}{l} run\_instr\ (\textbf{\textit{Invoke}}\ n_s\ n_e\ M')\ P\ h\ x\ reg\ C\ M\ ts\ pc\ frs\ r \\ = (let\ v = reg!n_s; n = n_e - n_s + 1; xp' = if\ v = Null\ then \lfloor NullPtrExcp \rfloor\ else\ None; \\ \quad C' = fst(the(h(the\_Addr\ v))); ns = upt\ n_s\ (n_e + 1); \\ \quad vs = regFetch\ reg\ (tl\ ns); ts' = map\ (the \circ typeof_h)\ vs; \\ \quad (D, T, regs_0, ins, xt) = method\ P\ C'\ M'\ ts'; reg' = replicate\ regs_0\ undefined; \\ \quad reg' = reg'@(v \# vs);\ f' = (Unit, reg', D, M', ts', 0) \\ \quad in\ (xp', h, f' \# (x, reg, C, M, ts, pc) \# frs, r)) \end{array} \quad (6)$$

The arguments are stored in registers whose beginning and ending indices are n_s and n_e. The first argument is the object reference as this-pointer. The execution first gets the object reference *v*. If *v* is *Null*, then a *NullPointer* exception is thrown. Otherwise, the function *regFetch* retrieves the arguments stored in registers excluding *v*, and the function composition *the* ∘ $typeof_h$ obtains their types *ts'*, and the *method* looks up the corresponding method with a return type t and a method body ($regs_0$, *ins*,*xt*). A new frame for the invoked method is created. The last registers of the new frame stores the arguments *v* and *vs*. The rest registers are initially filled with dummy values since their values are unknown at present. The *pc* of the new frame is set 0 and the just-caught exception is a dummy value *Unit*.

The whole definition is large but straightforward. The rule for the *MoveResult n* instruction updates the value of register *n* to the *r* which is updated by a return instruction. The rule for the instruction *MoveException n* updates the value of register *n* to the xcv which is updated when the function *lookupHandler* finds out an appropriate exception handler.

The function *exec* :: *dvm prog* ⇒ *dvm state* ⇒ *dvm state option* defines one-step execution. If the frame list is empty, then the execution will halt. The execution also halts if the current state is of the form (*Some xp*, *h*, *frs*, *r*) which means that an unhandled exception occurs. Otherwise, the function *instrs_of* retrieves the current instruction and the function *exec_instr* performs the current instruction:

$$\begin{array}{l} exec\ (P, xp, h, [], r) = None \\ exec\ (P, None, h, (x, reg, C, M, ts, pc)\#\ frs, r) = (let\ i = instr\_of\ \ P\ C\ M\ ts\ !\ pc; \\ \quad (xp, h', frs', r') = run\_instr\ i\ P\ h\ x\ reg\ C\ M\ ts\ r\ in \\ \quad \left| \begin{array}{l} case\ xp\ of\ \ None \Rightarrow (None, h', frs', r') \\ \lfloor a \rfloor \Rightarrow lookupHandler\ P\ a\ h\ ((x, reg, C, M, ts, pc)\#\ frs)\ r \end{array} \right|) \\ exec\ \ (P, \lfloor xp \rfloor, h, frs, r) = None. \end{array} \tag{7}$$

If an exception is raised, the function *lookupHandler* will look up the exception table, first in the current method, then in the context of the caller's frame recursively, continuing up the method invocation chain. If no suitable exception handler is found before the top of the method invocation chain is reached, the exception flag of the state remains to be set and the execution is terminated. If the function *lookupHandler* finds out a handler (*s*,*e*, *xc*,*h*), then the execution continues, setting the just-caught exception as the reference to the exception object, and pointing the program counter to *h*.

The above definition of the operational semantics of Micro-Dalvik is written in a functional rather than a relational style for direct executability. The one-step state transition relation can be defined in the inductively defined set, *exec*_1 *P*, and inductively defined predicate, *exec*_1 *P* σ σ' (written as $P \vdash \sigma \xrightarrow{dvm}_1 \sigma'$); the state transition for any finite number of steps is defined as the reflexive transitive closure of the one-step transition relation, written as $P \vdash \sigma \xrightarrow{dvm} \sigma'$ .The execution defined in this section does not perform any runtime type checks. It is the bytecode verifier that ensures that a Micro-Dalvik program is well-typed.

3. Type System for Micro-Dalvik

We take the approach of data flow analysis to solve the problem of bytecode verification where the program properties of interest are types. Klein and Nipkow define and verify a functional version of Kildall's algorithm [6], which is a standard data flow analysis tool. We use this general implementation framework: Each Micro-Dalvik instruction is viewed as a node of the control flow graph; the initial state type and the fixed-point iteration algorithm used to solve the general data flow problem are introduced in Section 4; the left is two main challenges – to specify the semilattice and transfer functions.

In the following we first give a concrete example of welltypings, and then specify the semilattice structure of Micro-Dalvik types, and give the stepwise construction process to obtain the semilattice for state type. Next we will define the transfer functions, and finally arrive at the definition of well-typings for a Micro-Dalvik program.

3.1. A Concrete Example

A state type characterizes a set of runtime states by giving type information. It is of the form (*xrT*, *rsT*) in our formalization. *xrT* and *rsT* are both lists whose elements are

types. *xrT* characterizes the just-caught exception object and the return value and *rsT* characterizes all values in registers. A state type represents the type information before execution of an instruction. It is a well-typing for an instruction if it satisfies both conditions:

- the instruction can be executed safely in a state whose just-caught exception object and the return value are typed according to *xrT* and whose registers are typed according to *rsT*, and
- it is consistent with the successors of the instruction.

The example in Table 1 shows the *pc*, instructions and corresponding state types. The method type is the full right hand side of the table. Register 5 and register 6 store arguments. We omit their types since they are both invariant during the execution. We here give the whole state type for the *0th* instruction:

([Err, Err], [Err, Err, Err, Err, Err,OK(Class B),OK(Class A)]).

It characterizes all states whose return value and just-caught exception object are unusable at present, thus [*Err*, *Err*], and whose register 5 contains an reference to an object of class *B* or to a subclass of B and whose register 6 contains an reference to an object of class *A* or to a subclass of *A*. The values in the register 0 through register 4 are also unusable at present, thus their types are all *Err*. We assume that class *B* is a subclass of *A*. The methods *create* and *createB* create objects of type *Class A* and *Class B*, respectively. We write *Cls B* instead of *OK*(*Class B*) for saving space, and similarly, *E* instead of *Err*, *I* instead of *OK Integer*, *Bool* instead of *OK Boolean*.

Table 1. A Concrete Example of Well-typings

pc	Instructions	State types
0	Move 0 5	([E,E],[E,E,E,E,E])
1	Move 1 6	([E,E],[Cls B,E,E,E,E])
2	Move 3 1	([E,E],[Cls B,Cls A,E,E,E])
3	Cmpeq 3 3 Null	([E,E],[Cls B,Cls A,E,Cls A,E])
4	IfFalse 3 9	([E,E],[Cls B,Cls A,E,Bool,E])
5	Move 3 0	([E,E],[Cls B,Cls A,E,Bool,E])
6	Invoke 3 3 "createB"	([E,E],[Cls B,Cls A,E,Cls B,E])
7	MoveResult 3	([E,Cls B],[Cls B,Cls A,E,Cls B,E])
8	Move 1 3	[E,E],[Cls B,Cls A,E,Cls B,E])
9	Const 3 (Intg 0)	([E,E],[Cls B,Cls A,E,E,E])
10	Move 2 3	([E,E],[Cls B,Cls A,E,I,E])
11	Move 3 2	([E,E],[Cls B,Cls A,I,E,E])
12	Const 4 (Intg 5)	([E,E],[Cls B,Cls A,I,I,E])
13	Cmplt 3 4 4	([E,E],[Cls B,Cls A,I,I,I])
14	IfFalse 3 21	([E,E],[Cls B,Cls A,I,Bool,I])
15	Move 3 0	([E,E],[Cls B,Cls A,I,Bool,I])
16	Invoke 3 3 "createA"	([E,E],[Cls B,Cls A,I,Cls B,I])
17	MoveResult 3	([E,Cls A],[Cls B,Cls A,I,Cls B,I])
18	Move 1 3	([E,E],[Cls B,Cls A,I,Cls A,I])
19	AriBinop Add 2 2 (Intg 1)	([E,E],[Cls B,Cls A,I,Cls A,I])
20	Goto 11	([E,E],[Cls B,Cls A,I,Cls A,I])
21	ReturnVoid	([E,E],[Cls B,Cls A,I,I,I])

A storage location may change its type during execution. Since there exist different execution paths to reach an instruction, the join operation, the common supertype of

the type information of these different paths has to be computed. Therefore, the type of a storage location can be *Err* if different types of this storage location are incompatible. This more general type information is less precise but it is still correct.

Table 1 is an example of a well-typed method. The types of register 1 and register 3 are more general types in the 9th entry. The reason is as follows. The predecessor of the 9th instruction *Const* 3 (*Intg* 0) may have either been *IfFalse* 3 9 (pc = 4) or Move 1 3 (pc = 8), so the type of the first register is either *A* or *B*; the common supertype is *A* since class *A* is the parent class of Class *B*. The type of the third register is either *Boolean* or *Class* B, which are incompatible; the only common supertype is *Err*. Similarly, the predecessor of *Move* 3 2 instruction (pc = 11) may have either been instruction *Move* 2 3 (pc = 10) or *Goto* 11 (pc = 20), so the type of the third register in the 11th entry is either Integer or Class A, the only supertype is *Err*.If we change the 6th instruction from *Invoke* 3 4 "*createB*" to *MoveResult* 3, the method will not be well-typed. The reason is that *MoveResult* 3 would try to take an unusable value, and could therefore not be executed.

3.2. A Basic Semilattice Structure

Our semilattice construction is based on the supremum semilattice described in [3]. A triple (*A*,*r*,*f*) of type *'a sl* = *'a set* × *'a ord* × *'a binop* is by definition a semilattice iff the predicate *semilat* :: *'a sl* ⇒ *bool* holds:

$$\begin{aligned} semilat\ (A,r,f) \equiv\ & order\ r \wedge closed\ A\ f \\ & \wedge (\forall x \in A.\ y \in A.\ x\ \hat{o}_r\ x\ \grave{o}_f\ y) \wedge (\forall y \in A.\ y \in A.\ y\ \hat{o}_r\ x\ \grave{o}_f\ y) \\ & \wedge (\forall x \in A.\ \forall y \in A.\ \forall z \in A.\ x\ \hat{o}_r\ z \wedge y\ \hat{o}_r\ z \longrightarrow x\ \grave{o}_f\ y\ \hat{o}_r\ z. \end{aligned} \tag{8}$$

The notations $x\ \hat{o}_r\ y$ and $x\ x\ \grave{o}_f\ y$ are $r\ x\ y$ and $f\ x\ y$, respectively. The corresponding type synonym definitions are *'a ord* = *'a* ⇒*'a* ⇒*bool* and *'a binop* = *'a*⇒*'a* ⇒*'a*.

In terms of data flow analysis and type system the set *A* is a set of types, the order *r* is the subtype relation and the *f* is the supremum which computes the least common supertype of two types. To model the situation where the supremum of two types does not exist, a new type is introduced: *'a err* = Err |OK *'a*; We define *err A* =*Err* ∪ *OK x*|*x*∈*A*. Therefore, we have a new type synonym: *'a ebinop* = *'a*⇒*'a* ⇒*'a err*. An ordering on *'a err* is based on an existing ordering on *'a* using the definition and the definition *lift* is also useful for the semilattice construction:

$$\begin{aligned} le\ r\ e_1\ e_2 &= (case\ e_2\ of\ Err \Rightarrow True \mid OK\ y \Rightarrow (case\ e_1\ of\ Err \Rightarrow False \mid OK\ x \Rightarrow r\ x\ y)) \\ lift\ f\ e_1\ e_2 &= (case\ e_1\ of\ Err \Rightarrow Err \mid OK\ x \Rightarrow (case\ e_2\ of\ Err \Rightarrow Err \mid OK\ y \Rightarrow f\ x\ y)) \end{aligned} \tag{9}$$

A triple L of type 'a esl = 'a set × 'a ord × 'a ebinop is an err-semilattice iff the predicate err_semilat :: 'a esl ⇒ bool holds:

$$err\_semilat\ L == semilat\ (sl_e\ L). \tag{10}$$

where sl_e converses 'a esl into *'a err sl*:

$$sl_e\ (A,r,f) \equiv (err\ A, le\ r, lift\ f). \tag{11}$$

The Micro-Dalvik operates on the type ty which is either a primitive type *pt* or a reference type *rt*. A *pt* is either an Integer or a Boolean, and a *rt* is either a class type *Class cname* or a null type *NT* .

The constant definition *esl* turns types supported by a Micro-Dalvik program into an err-semilattice, defined in the theory file *Semitype.thy*, *esl P* ≡ (*types P*, *subtype P*, *sup P*), where *types P* is the set of all types declared in the program *P*, *subtype P* is the

standard subtype ordering. The subtype relation is a widening relation of the subclass relation, written $P \vdash T \leq T'$:

$$P \vdash T \leq T;\ P \vdash NT \leq Class\ C;\ \frac{P \vdash C \circ^{*} D}{P \vdash Class\ C \leq Class\ D}. \tag{12}$$

The subtype relation means that any type T is a subtype of itself and NT is the subtype of any reference type. If a class C is a subclass of D, then $Class\ C$ is the subtype of $Class\ D$.

The *sup P* defines the supremum operations of two elements of type *ty* (shown in Table 2; we omit OK for non-Err results). The *X* in the last cell could be *C* if *C* is the superclass of *D*, or *D* if *D* is the superclass of *C*, or *Object* if there exists no subclass relation between *C* and *D*.

Table 2. The Supremum Operations on Micro-Dalvik Types

	Void	Boolean	Integer	NT	Class D
Void	Void	Err	Err	Err	Err
Boolean	Err	Boolean	Err	Err	Err
Integer	Err	Err	Integer	Err	Err
NT	Err	Err	Err	NT	Class D
Class C	Err	Err	Err	Class X	Class X

Next we introduce a predicate *wf_prog* :: *'m wf_mdecl_test* ⇒ *'m prog* ⇒ *bool* used for well-formedness which is a prerequisite for most theorems:

$$\begin{aligned}
&wf_prog\ wf_md\ P \equiv wf_syscls\ P \wedge distinct_fst\ P \wedge (\forall c \in set\ P.\ wf_cdecl\ wf_md\ P\ c)\\
&wf_cdecl\ wf_md\ P \equiv \lambda(C,(D, fs, ms)).\ (\forall f \in set\ fs.\ wf_fdecl\ P\ f) \wedge distinct_fst \wedge\\
&(\forall m \in set\ ms.\ wf_mdecl\ wf_md\ P\ C\ m) \wedge\\
&distinct_fst\ (\lambda(mName, ts, t, mb).\ ((mName, ts), t, mb))\ ms) \wedge\\
&(C \neq Object \longrightarrow is_class\ P\ D \wedge \neg P \vdash D \circ^{*} C \wedge\\
&\quad \forall (M, ts, t, m) \in set\ ms.\ \forall D'\ t'\ m'.\ seesMethod\ P\ D\ M\ ts\ t'\ m'\ D' \longrightarrow P \vdash t \leq t'.
\end{aligned} \tag{13}$$

The well-formedness definition on programs is parameterized by a well-formedness test both for the methods of Jinja and the JVM programs, which is an embodiment of the unified model described in [3]. We will use it for the well-formedness of the Micro-Dalvik programs in Section 3.5. At the present, we do not need to discuss the well-formedness of a method body.

The generic constraints on programs are almost same. A program is well-formed iff it contains all system classes, all class declarations are well-formed and no class is declared twice. A class declaration is well-formed iff all field declarations and method declarations are well-formed, no field or method is declared twice, and if *C*≠*Object*, then its superclass *D* is not a subclass of *C*. The difference is that method override involves both the method name and parameter types, instead of only the method name. If a method declared in *C* overrides a method declaration visible from its direct superclass *D*, then the new declaration must have more specific result type.

Now we can prove the following theorem.

Theorem 3.1. *SemiType.esl P* is an err-semilattice and there is no infinite ascending chain in *types P* under the condition that the Micro-Dalvik program *P* is well-formed.

The proof is as follows. The *subtype P* is obviously reflexive and transitive. The well-formedness of a Micro-Dalvik program requires that a non-Object class *C* has its super class *D* and *D* is not a subclass of *C*, thus suptype P is antisymmetric, and therefore a partial order. It is easy to see that sup P is closed with respect to types *P*. The order relation *suptype P* satisfies the ascending chain condition on types *P* since a

well-formed program has a class hierarchy where *Object* is at the top. Hence, *SemiType.esl P* is an err-semilattice for *ty*. The whole proof scripts are almost same with those in [3], except some differences in details.

In the following we write *SemiType.esl P* as esl_P.

3.3. Semilattice Construction for Micro-Dalvik's State Type

3.3.1. State Type of Micro-Dalvik: The term state type characterizes the type information of the state before execution of an instruction. As discussed in Section 2.2.1 where we need to consider two special values passed through the interpreter, we formalize the state type of Micro-Dalvik as a tuple (*xrT*,*rsT*) of type:

$$ty_i = ty_r \times ty_r \text{ where } ty_r = ty\ err\ list. \tag{14}$$

The component *rsT* is a list of types corresponding to the register values. The component is a list of size two. The first represents the type of the type of just-caught exception and the second represents the type of the return value. These types may be unknown at some execution point, and thus are formalized as *ty err* instead of *ty*.

Data flow analysis needs to indicate the state type of instructions that have not been reached yet, encoded by *None*, and thus we have $ty_i' = ty_i\ option$. A method type is of type $ty_m = ty_i'\ list$.

Finally, to indicate type errors during type inference, an additional *Err* layer needs to be added. Therefore, we arrive at the state type $ty_i'\ err$.

3.3.2. Semilattice for State Type: In this section we turn $ty_i'\ err$ into a semilattice, following an approach of lifting an existing semilattice into a new semilattice. In the previous section we have already obtained an err-semilattice esl_P. Starting with it, we give the following construction steps to obtain the semilattice structure of $ty_i'\ err$.

- $sl_e :: 'a\ esl \Rightarrow 'a\ err\ sl$. The definition sl_e can form a semilattice for *ty err* form esl_p. Since esl_p is an err-semilattice, and *err_semilat L* is an abbreviation of *semilat* (sl_e *L*), $sl_e\ esl_p$ is a semilattice with top element *Err*, and there is no ascending chain in the set of this new semilattice.
- $sl_l :: nat \Rightarrow 'a\ sl \Rightarrow 'a\ list\ sl$. The definition sl_l turns a semilattice for *ty* into a semilattice for ty_r:

$$\begin{aligned} &sl_l\ n \equiv \lambda(A, r, f).\ (list\ n\ A, le_l\ r, sup_l\ f) \\ &list\ n\ A \equiv \{xs.\ size\ xs = n \wedge set\ xs \subseteq A\} \\ &le_l\ r \equiv list\_all2\ (\lambda x\ y.\ x\ \hat{o}_r\ y) \\ &sup_l\ f \equiv (\lambda xs\ ys.\ map\ (split\ f)\ (zip\ xs\ ys)). \end{aligned}$$

The *list_all2*, *map*, *split* and *zip* are standard functions predefined in Isabelle/HOL. The ordering relation of two lists *xs* and *ys*, $xs\ \hat{o}_{le_l}\ ys$, is written $xs[\hat{o}_r]ys$. By the definition of sl_l, the semilattice structure of type ty_r is built where each element in the set is a bounded list whose elements are taken from the existing set; the ordering of two lists with different lengths is *False*, or it is defined based on the existing ordering of each corresponding element from these two lists; the supremum is a list where each element is the existing supremum of each corresponding element from two lists. Therefore, we can build a new semilattice structure $sl_l\ n\ (sl_e\ esl_p)$ with a fixed length *n*. This structure can be proved to be a semilattice and satisfies no infinite ascending chain condition if we can prove the following lemma:

Lemma 3.2. If L is an semilattice and there is no infinite ascending chain in r, then $(sl_{\mathrm{l}}\ n\ L)$is also a semilattice, and there is no infinite ascending chain in le_{l}.

$$semilat\ L = \text{Þ}\ \ semilat(sl_l\ n\ L)$$
$$order; acc\ r\ = \text{Þ}\ \ acc(le_l\ r)$$

Proof. First we need to prove $order\ \ r = \text{Þ}\ \ order(le_{\mathrm{l}}\ r)$. The proof is straightforward. The *list_all2* is reflexive, antisymmetric, and transitive, and thus order($le_{\mathrm{l}}\ r$) is true. Next it also needs to prove $closed\ A\ f\ = \text{Þ}\ \ closed$ (*list n A*) ($sup_l\ f$). The proof first uses the definition of *closed*, and then is by induct on the length of the list. Finally three other properties required by a semilattice are proved by a series of auxiliary lemmas which are proved using *unfold* and *simp* proof methods. Therefore, we can prove that $sl_{\mathrm{l}}\ n\ (sl_{\mathrm{e}}\ esl_{\mathrm{p}})$ is a semilattice. The proof for no ascending chain is a little complex. It use an equivalent concept of well-founded relation which is defined in the theory file *Wellfounded.thy* in Isabelle/HOL.

- sl_{p} :: *'a sl* $\Rightarrow$*'b sl* $\Rightarrow$(*'a* $\times$*'b*) *sl*. The definition sl_{p} turns a semilattice for ty_{r} into a semilattice for ty_{i}:

$$sl_p \equiv \lambda(A, r_1, f_2)\ \ (B, r_1, f_2).\ \ (A \times B, le_p\ r_1, r_2, sup_P\ \ f_1\ \ f_2)$$
$$le_P\ \ r_1\ r_2 \equiv \lambda(a_1, b_1)\ \ (a_2, b_2).\ \ a_1\ \hat{o}_{r_1} a_2 \wedge b_1\ \hat{o}_{r_2} b_2$$
$$sup_P\ \ f\ \ g \equiv \lambda(a_1, b_1)\ \ (a_2, b_2).\ \ Pair\ (a_1\ \grave{o}_f a_2)\ \ (b_1\ \grave{o}_g b_2)$$

To lift a semilattice for ty_{r} to a semilattice for ty_{i}, we use the operation $\times$ and the new ordering and new supremum are defined based on the existing ordering and supremum of each component. Now we obtainthis new structure $sl_{\mathrm{p}}\ L_1\ L_2$ which are both semilattices for type ty_{r}, and L_1 is $sl_{\mathrm{l}}\ n_1\ (sl_{\mathrm{e}}\ esl_{\mathrm{p}})$, and L_2 is $sl_{\mathrm{l}}\ n_2\ (sl_{\mathrm{e}}\ esl_{\mathrm{p}})$. This new structure is proved to be a semilattice and it satisfies no ascending chain condition if we can prove the following lemma:

Lemma 3.3. If L_1 and L_2 are both semilattices and there is no infinite ascending chain in r_{A} and in r_{B}, then $sl_{\mathrm{p}}\ L_1\ L_2$ is also a semilattice, and there is no infinite ascending chain in $le_{\mathrm{p}}\ r_{\mathrm{A}}\ r_{\mathrm{B}}$.

$$\forall L_1\ L_1.\ \ semialt\ L_1; semilat\ L_2\ = \text{Þ}\ \ semilat(sl_p\ L_1\ \ L_2)$$
$$acc\ r_A; acc\ r_{\mathrm{B}}\ = \text{Þ}\ \ acc(le_p\ r_A\ r_B)$$

- sl_{o} :: *'a sl* $\Rightarrow$*'a option sl*. The definition sl_{o} can turn a semilattice for ty_{i} into a semilattice for ty_{i}':

$$sl_o \equiv \lambda(A, r_A, f_A)\ \ (opt\ A, le_o\ r, sup_o\ \ f)$$
$$le_o\ r\ o_1\ o_2 \equiv case\ o_2\ of\ \ None \Rightarrow o_1 = None \mid \lfloor y \rfloor \Rightarrow (case\ o_1\ of\ \ None \Rightarrow True\ \mid \lfloor x \rfloor \Rightarrow x\ \hat{o}_r y$$
$$sup_o\ \ f\ o_1\ o_2 \equiv case\ o_1\ of\ \ None \Rightarrow o_2 \mid \lfloor x \rfloor \Rightarrow (case\ o_2\ of\ \ None \Rightarrow o_1 \mid \lfloor y \rfloor \Rightarrow \lfloor (f\ x\ y) \rfloor$$

Firstly, we insert an additional element None into the existing set and make None the bottom element of the exsiting ordering, and then define the supremum of and any element is the latter, and finally let non-bottom elements behave like the existing ordering and supremum we obtain a tuple of type ty_{i}'. Now we obtain this new structure $sl_{\mathrm{o}}\ (sl_{\mathrm{p}}\ L_1\ L_2)$, which can be proved to be a semilattice and it satisfies no infinite ascending condition using the theorem below:

Lemma 3.4. If L is a semilattices and there is no infinite ascending chain in r, then is also a semilattice, and there is no infinite ascending chain in $le_{\mathrm{o}}\ r$.

$$semilat\ L \Rightarrow semilat(sl_o\ L)$$
$$acc\ r \Rightarrow acc(le_o\ r)$$

- $esl_e :: 'a\ sl \Rightarrow 'a\ esl$. The definition esl_e can turn a semilattice for ty_i' into an err-semilattice. The new set and ordering are the same and the new supremum is defined with $\lambda x\ y.\ OK(f\ x\ y)$ as:

$$esl_e \equiv \lambda(A, r_A, f_A)\ (A, r, \lambda x\ y.\ OK(f\ x\ y))$$

Now we obtain this new structure $esl_e\ (sl_o\ (sl_p\ L_1\ L_2))$, which can be proved to be an err-semilattice using the theorem below:

Lemma 3.5. If L is a semilattices, then $esl_e\ L$ is an err-semilattice.

$$semilat\ L \Rightarrow err\_semilat(esl_e\ L)$$

Therefore, given a Micro-Dalvik program P and a bound n we can finally obtain its semilattice for $ty_i'\ err$ by the function sl_d:

$$sl_d\ P\ n \equiv sl_e\ (esl_e\ (sl_p\ (r\_sl\ P\ (Suc\ (Suc\ 0))\ (r\_sl\ P\ n))))\ \text{and}\ r\_sl\ P\ n \equiv sl_l\ n\ (sl_e\ n\ esl_P)$$

The process of building semilattice is different from what is described in [3]. The reason for the difference is the absence of the operand stack. The values on the operand stack must be of a known type. In Micro-Dalvik, both *xrT* and *rsT* may contain an unknown type.

Theorem 3.6. If P is well-formed, $sl_d\ P\ n$ is a semilattice for ty_i' err and the ordering satisfies the ascending chain condition.

$$wf\_prog\ wf\_mb\ P \Rightarrow semilat(sl_d\ P\ n)$$
$$wf\_prog\ wf\_mb\ P \Rightarrow acc(fst(snd(sl_d\ P\ n)))$$

Proof. Having Lemma 3.2 through 3.5 we can prove this theorem easily. The proof is straightforward. The well-formedness ensures that esl_p is an err-semilattice using Theorem 3.1. Using lemma 3.2 through lemma 3.5 we can ensure that $sl_d\ P\ n$ is a semilattice since the new structure of each lifting from a semilattice structure has been proved to be a semilattice or an err-semilattice and the new ordering satisfies the ascending chain condition.

3.4. Transfer functions of Micro-Dalvik

In this section we employ the semilattice structure $sl_d\ P\ n$ to formalize the two conditions on a well-typing described in Section 3.1. The main task is to define the applicability before executing an instruction and to define the effect after execution of an instruction.

3.4.1. Applicability before execution: We start with describing the applicability in the normal case which defines the conditions each instruction requires. The core function app_i means that for a method with a return type t_r in a Micro-Dalvik program *P*, an instruction with the program counter *pc* is applicable in the current state type (*xrT*,*rsT*).

We look at the applicability rule for the instruction *Invoke*:

$$app_i\ (Invoke\ n_s\ n_e\ M', P, pc, t_r, (xrT, rsT)) = (n_s < |rsT| \wedge n_e < |rsT| \wedge (ok\_val\ (regT!n_s) \neq NT$$
$$\longrightarrow (\exists C\ D\ ts\ t\ mb.\ ok\_val\ (rsT!n_s) = \text{Class}\ C \wedge seesMethod\ P\ C\ \text{M}'\ ts\ t\ mb\ D \wedge$$
$$P \vdash (map\ ok\_val\ (regFetch\ regT\ (tl\ (upt\ ns\ (ne+1))))) [\leq] ts)))$$

(15)

This rule ensures that n_s and n_e is less than the number of registers; If the object reference is not *Null*, then the method *M* whose parameter types are *ts* is visible in some class, scanning the class hierarchy upwards starting from *C* which is the type of the object reference.

The applicability for exceptional execution *xp_app* requires that if an instruction is within the interval [*s, e*] of an exception entry (*s,e,xC,h*), then the relevant exception type of a particular instruction is a subtype of *Class xC* and *xC* is the name of an exception class defined in the program. Additionally, an unreachable instruction is considered applicable, and thus, the whole applicability for execution is defined by the function *app* :: $instr \Rightarrow 'm\ prog \Rightarrow ty \Rightarrow pc \Rightarrow nat \Rightarrow ex\_table \Rightarrow ty_i' \Rightarrow bool$:

$$app\ i\ P\ t_r\ pc\ mpc\ xt\ st = case\ st\ of\ None \Rightarrow True \mid \lfloor \tau \rfloor \Rightarrow app_i\ (i, P, pc, t_r, \tau) \wedge xp\_app\ i\ P\ pc\ xt\ \tau \wedge (\forall (q, \varepsilon) \in set\ (eff\ i\ P\ pc\ xt\ st).\ q < mpc)). \quad (16)$$

It requires that an instruction *pc* of a method in a program *P* is applicable iff both app_i and *xp_app* are *true*, and all the successors of this instruction are within this method (mpc is the length of the instruction sequence). Next we discuss the effect defined by the function *eff*.

3.4.2. Effect after Execution: In this section we introduce the effect of execution of Micro-Dalvik programs. First the execution of a Micro-Dalvik instruction is defined by *eff* :: $instr \Rightarrow 'm\ prog \Rightarrow pc \Rightarrow ex\_table \Rightarrow ty_i' \Rightarrow (pc \times ty_i')list$:

$$eff\ i\ P\ pc\ xt\ st = case\ st\ of\ None \Rightarrow [] \mid \lfloor \tau \rfloor \Rightarrow (norm\_eff\ i\ P\ pc\ \tau)@(xp\_eff\ i\ P\ pc\ \tau\ xt)$$

( )

This means that the whole execution effect is joining the normal case and exceptional case.

The function *norm_eff* defines the normal execution:

$$norm\_eff\ i\ P\ pc\ \tau = map\ (\lambda pc'.\ (pc', \lfloor eff_i\ (i, P, \tau) \rfloor))\ (succs\ i\ \tau\ pc) \quad (17)$$

The execution effect considers the successor instructions for the sake of the iteration algorithm needed by the bytecode verifier. The successor of most instructions is just plus 1. The conditional branch instruction has two successors, one plus 1 and the other plus the specified offset. The *Return* instructions have no successors since bytecode verification works within the single method. The *Throw* instruction has no successor for a normal execution either. Additionally, the effect of an unreachable instruction is *empty*.

The core function eff_i means that executing an instruction in a state type arrives at a new state type after execution. We look at the definition of the effect for the instruction *Invoke*:

$$\begin{aligned} &eff_i\ (Invoke\ n_s\ n_e\ M', P, (xrt, rsT)) = \\ &let\ oT = rsT\ !\ n_s; C = (case\ oT\ of\ (OK\ TC) \Rightarrow (the\_class\ TC)); \\ &ns = upt\ n_s\ (n_e + 1); ts = map\ ok\_val\ (regFetch\ rsT\ n_s); (D, t_r, mb) = method\ P\ C\ M'\ ts\ in \\ &((hd\ (xrT)\#((OK\ t_r)\#[]), rsT)) \end{aligned} \quad (18)$$

Given the method name and types of parameters, the function *method* looks up the method with a return type t_r and the method body *mb* visible in *D*, scanning the class hierarchy upwards starting from *C*. Therefore, after the execution of the instruction the return type is t_r and the others keep the same.

The function *xp_eff* defines the effect for exceptional execution:

$$xp\_eff\ i\ P\ pc\ \tau\ xt = (let\ (xrT, rsT) = \tau\ in\ (map\ (\lambda(s,e,xC,h).\ (h, \lfloor ((OK\ (Class\ xC))\#(tl\ xrT), rsT) \rfloor))\ (relevant\_entries\ P\ i\ pc\ xt)) \tag{19}$$

When an exception is thrown, the type of the exception object is stored in the first element of *xrT*. The program jumps to the instruction at the position *h* which is the exception handler.

3.5. Well-typings of Micro-Dalvik

We already give a concrete example of well-typings in Section 3.1. In this section we define well-typings for the Micro-Dalvik. First we show how the well-typings τs:: ty_m fit the instruction sequence of a method.

The predicate $P,t_r,mpc,xt \vdash i, pc :: \tau s$ is defined with respect to *app* and *eff*. It means that an instruction *i* with *pc* is applicable in the state type $\tau s!\ pc$, and the state types of all its successors is not greater than the corresponding state type in τs, which means that all its successors are consistent with the expected:

$$P,t_r,mpc,xt \vdash i, pc :: \tau s \equiv app\ i\ P\ t_r\ \ pc\ mpc\ xt\ (\tau s!\ pc) \wedge \forall (pc', \tau') \in set\ (eff\ i\ pc\ xt\ (\tau s!\ pc)).\ P \vdash \tau' \preceq \tau s!\ pc'). \tag{20}$$

In order to obtain an executable bytecode verifier, a start condition and some side conditions are also defined in the following. The wt_start is below:

$$wt\_start\ P\ C\ ts\ mxr_0\ \tau s \equiv P \vdash \left\lfloor \begin{matrix} ([Err, Err], (replicate\ mxr_0\ Err)@ \\ (OK\ (Class\ C)\#(map\ OK\ ts))) \end{matrix} \right\rfloor \preceq \tau s!0) \tag{21}$$

It describes the correct state type for instruction 0. The last *N* registers store arguments, and thus are already known type. The left registers do not have a value yet, and thus are of unknown type. The return value and the just-caught exception at this point are both unknown type. Thus they are encoded by *Err*.

Additionally, the well-typings require that the method type only contains declared classes: $check\_types\ P\ mxr\ \tau s \equiv set\ \tau s \subseteq states\ P\ mxr$ where $states\ P\ mxr \equiv fst\ (sl_d\ P\ mxr)$. Now the predicate *wt_method* can be defined and it decides whether a method is well-typed:

$$\begin{aligned} &wt\_method\ P\ C\ ts\ t_r\ mxr_0\ is\ xt\ \tau s \equiv size\ is > 0 \wedge size\ \tau s = size\ is \wedge \\ &(let\ mxr = getmaxr\ is - size\ ts\ in\ (\forall pc < size\ is.\ P,t_r,mpc,xt \vdash i, pc :: \tau s) \wedge \\ &\quad wt\_start\ P\ C\ ts\ mxr\ \tau s \wedge\ check\_types\ P\ (getmaxr\ is)\ (map\ OK\ \tau s) \wedge \\ &\quad (\_, \_, \_, h) \in xt.\ \exists n.\ is!h = (MoveException) \wedge \\ &\quad \exists n > 0.\ is!n = (MoveResult\ n) \longrightarrow \exists n_s\ n_e\ M.\ is!(n-1) = (Invoke\ n_s\ n_e\ M)) \end{aligned} \tag{22}$$

It says that if τs covers all instructions and only contains valid types in *P*, the method calling convention is respected such that the method has a correct state type for instruction 0, all instructions are well-typed, and the first instruction in each exception handler is *MoveException* instruction, and the instruction *MoveResult* only appears immediately after an appropriate *Invoke* instruction, then τs are well-typings and this method is well-typed.

Now we can define the well-typings of Micro-Dalvik programs. First we give a type synonym ty_p for the Micro-Dalvik program type: $ty_p = cname \Rightarrow mname \Rightarrow ty\ list \Rightarrow ty_m$. It computes a list of elements which are either state types or *None*, given a method name and its parameter types and the class name. A well-typed Micro-Dalvik program requires that for each method *M* with parameter types ts in a class C defined in this program, there exists Φ such that $\Phi\ C\ M\ ts$ is well-typings. Therefore, the well-formedness of a Micro-Dalvik program is defined by *wf_dvm_prog*: :$dvm\_prog \Rightarrow bool$:

$$wf\_dvm\_prog_{\Phi} \equiv$$
$$wf\_prog\ (\lambda P\ C\ (M, ts, t, (r_0, is, xt)).\ wt\_method\ P\ C\ ts\ t_r\ mxr_0\ is\ xt\ (\Phi\ C\ M\ ts)) \tag{23}$$

The predicate *wf_prog* is a general framework described in [3] both for Java and JVM programs. We use it for the Micro-Dalvik program's well-formedness by giving the argument which is used to describe what the well-typedness of a Micro-Dalvik method is.

4. Type Inference and Type Safety

Type inference is a process of computing a method type from the instruction sequence of a method. We use an functional implement of Kildall's algorithm *kildall* for type inference. We fill this algorithm with concrete ordering *r*, supremum operations *f* and flow functions defined with *app* and *eff* as parameters defined in section 3, and thus obtain a Micro-Dalvik version of the Kildall's algorithm, *kildvm*:

$$\begin{aligned} &kildvm\ P\ mxr\ t_r\ is\ xt \equiv \\ &kildall\ (\ fst(snd\ sl_d\ P\ mxl))\ (snd(snd\ sl_d\ P\ mxl))\ (exec\ P\ t_r\ xt\ is)) \end{aligned} \tag{24}$$

where

$$\begin{aligned} &exec\ P\ t_r\ xt\ is \equiv err\_step\ |is|\ (\lambda pc.\ app\ (is!pc)\ P\ t_r\ pc\ |is|\ xt)\ (\lambda pc.\ eff\ (is!pc)\ P\ pc\ xt) \\ &err\_step\ n\ app\ eff\ pc\ \varphi \equiv case\ t\ of\ Err \Rightarrow error\ n\ | \\ &OK\ \varphi \Rightarrow if\ app\ p\ \varphi\ then\ map\_snd\ OK\ (eff\ p\ \varphi)\ else\ error\ n \end{aligned} \tag{25}$$

The function *err_step* works on type $ty_i'\ err$ instead of ty_i since type inference needs to indicate type errors during the execution. If the current state type is *Err* or the current instruction is not applicable in this state type, then the function *err_step* will propagate the error element *Err* to every position in the method type, which means that a type error happens in the Micro-Dalvik program.

An executable bytecode verifier for Micro-Dalivk methods is obtained by giving the correct start value and filling the state type at positions greater than 0 with *OK None*. Finally, we obtain the executable bytecode verifier, *wt_kidall* :: *dvm_prog* $\Rightarrow$ *cname* $\Rightarrow$ *ty list* $\Rightarrow$ *ty* $\Rightarrow$ *nat* $\Rightarrow$ *instr list* $\Rightarrow$ *ex_table* $\Rightarrow$ *bool*:

$$\begin{aligned} &wt\_kildall\ P\ C\ ts\ t_r\ mxr_0\ is\ xt = 0 < size\ is \wedge \\ &(let\ \ first = \lfloor ([Err, Err], (replicate\ mxr_0\ Err) @ (OK\ (Class\ C) \# (map\ OK\ ts))) \rfloor; \\ &\quad start = OK\ \ first \# (replicate\ (size\ is - 1)\ (OK\ None)); \\ &\quad result = kildvm\ P\ (1 + size\ ts + mxr_0)\ t_r\ is\ xt\ start \\ &in\ \forall n < size\ is.\ \ result!n \neq Err). \end{aligned} \tag{26}$$

Now we can lift to the full Micro-Dalvik programs:

$$\begin{aligned} &wf\_dvm\_prog_k\ P \equiv wf\_prog\ (\lambda P\ C\ (M, ts, t, (mxr_0, is, xt)). \\ &wt\_kildall\ P\ C\ ts\ t\ (mxr_0\ is\ xt)\ P \end{aligned} \tag{27}$$

It can be proved that the executable bytecode verifier is sound and recognizes all well-typed programing: $wf\_dvm\_prog_k\ P = wf\_dvm\_prog\ P$.

Now we prove that well-typed Micro-Dalvik programs will never return a type error. We can thus can prove that the bytecode verifier is correct and it guarantees type safety of Micro-Dalvik programs.

First we need a formalization to indicate runtime type error. Thus, a new type synonym is defined: datatype *'a type_error = TypeError | Normal 'a*. Next a single-step defensive execution is defined based on the operational semantics defined in Section 2:

exec_d ≡ *if check P σ then Normal* (*exec* (*P*, *σ*)) *else TypeError*. The core function *check*$_i$ called by the function *check* is parallel to the definition of *app*$_i$ in Section 3; it works on values instead of on types. This is called defensive execution because it performs runtime checks. The execution continues only when the function check return *true*, or it produces a *TypeError*. Using the same way as discussed in Section 2, we can define its relational style as: $P \vdash \sigma \xrightarrow{ddvm}_1 \sigma'$ and $P \vdash \sigma \xrightarrow{ddvm} \sigma'$

Assuming the defensive execution starts by invoking an existing method *M* without parameters in a class *C*, we give the following context:

local start = fixes P and C and M and σ and t and m
assumes wf: wf_dvm_prog P
assumes sees:seesMethod P C M [] t mb C
defines σ= start_state P C M [],

where

$$\begin{aligned} &start\_state\ P\ C\ M = (let\ (D, t, loc, ins, xt) = method\ P\ C\ M\ [];\\ &reg = replicate(maxR\ ins + 1)\ undefined\\ &in\ (None, start\_heap\ P, [(undefined, reg, C, M, [], 0)], undefined)). \end{aligned} \tag{28}$$

Now we can give the type safety theorem:

Theorem 4.1. (*in start*)

$$P \vdash Normal\ \sigma \xrightarrow{ddvm} \sigma' \Rightarrow \sigma' \neq TypeError$$

To prove this theorem, first we need to build a conformance relation between well-typedness and the normal machine state. An approach is introduced by Pusch [4]. It proves that if all values at run time conform to the types given in the well-typings, thenthere is no type error.

The definition *correct_state*, written as $P, \Phi \vdash \sigma \surd$ indicates that the machine state conforms to the program type. Its definition is:

$$\begin{aligned} &P, \Phi \vdash \sigma \surd \equiv \lambda(xp, h, frs, r).\ case\ xp\ of\ None \Rightarrow (case\ frs\ of\ [] \Rightarrow True\\ &| (fr \# frs) \Rightarrow P \vdash h \surd \wedge (let\ x, reg, C, M, ts, pc) = fr\\ &\quad in\ \exists t\ is\ \tau.\ seesMethod\ P\ C\ M\ ts\ t\ (n, is, xt)\ C \wedge\ (\Phi\ C\ M\ ts)!PC = \lfloor \tau \rfloor \wedge\\ &\quad conf\_fr\ P\ h\ \tau\ is\ fr\ r \wedge\ conf\_frs\ P\ h\ \Phi\ M\ ts\ t\ frs\ r)) | \lfloor x \rfloor \Rightarrow frs = [] \end{aligned} \tag{29}$$

where the predicate *conform_fr* defines the single-frame conformance relation:

$$\begin{aligned} &conf\_fr\ p\ h \equiv \lambda(xrT, rsT)\ is\ (x, reg, C, M, ts, pc)\ r.\\ &P, h \vdash x :\leq_T (hd\ xrT)\ \wedge\ P, h \vdash r :\leq_T hd(tl\ xrT) \wedge P, h \vdash reg\ [:\leq_T]\ rsT\ \wedge\ pc < size\ is \end{aligned} \tag{30}$$

It specifies that a single frame conforms if its just-caught exception, the return value and the register set conform and if the program counter lies inside the instruction list. The predicate *conform_frs* gives the conformance relation of the structure of frames, required by the proof for the *Invoke* instruction:

$$\begin{aligned} &conf\_frs\ p\ h\ \Phi\ M_0\ ts_0\ t_0\ []\ r = True\\ &conf\_frs\ p\ h\ \Phi\ M_0\ ts_0\ t_0\ (fr \# frs)\ r = (let\ (x, reg, C, M, ts, pc) = fr\ in\ (\exists xrT\ rsT\ t\ n\ is\ xt.\\ &\quad (\Phi\ C\ M\ ts)!pc = \lfloor (xrT, rsT) \rfloor \wedge (seesMethod\ P\ C\ M\ ts\ t\ (n, is, xt)\ C) \wedge\\ &\quad (\exists ns\ ne\ D\ t'\ m\ D'.\ is!pc = (Invoke\ n_s\ n_e\ M_0) \wedge ok\_val(rsT!ns) = Class\ D \wedge\\ &\quad (seesMethod\ P\ D\ M_0\ ts_0\ t'\ m\ D') \wedge P \vdash t_0 \leq t') \wedge\\ &\quad conf\_fr\ P\ h\ (xrT, rsT)\ is\ fr\ r \wedge conf\_frs\ P\ h\ \Phi\ M\ ts\ t\ frs\ r)) \end{aligned} \tag{31}$$

Now we give the auxiliary theorem which states that a single-step defensive execution do not produce type errors if it starts to execute in a conform state.

Theorem 4.2.

$$\begin{array}{l} \textit{fixes } \sigma :: \textit{dvm\_state} \quad \textit{assumes "wf\_dvm\_prog}_{\Phi}\ P\text{" and "}P,\Phi \vdash \sigma\ \surd\text{"} \\ \textit{shows exec\_d } P\ \sigma \neq \textit{TypeError} \end{array}$$

The proof is by case distinction on the current instruction in σ. The well-formedness ensures the applicability. The parallelism between the function app_i and eff_i ensures the function *check* returns true. Therefore, $exec\_d\ P\ \sigma \neq TypeError$.

Theorem 4.3.

$$wf\_dvm\_prog_{\Phi}\ P; seesMethod\ P\ C\ M\ [\,]\ t\ m\ C \Rightarrow P,\Phi \vdash start\_state\ P\ C\ M\ \surd$$

This theorem says that the starting state conforms. It is assumed that the Micro-Dalvik starts with a method *M* without parameters in the class *C* in the program *P*. It is proved by using the definitions of *correct_state* and *start_state*.

We also prove an invariance of the conformance relation in a well-typed program as theorem 4.4.

Theorem 4.4.

$$wf\_dvm\_prog_{\Phi}\ P; P \vdash \sigma \xrightarrow{dvm} \sigma' \Rightarrow P,\Phi \vdash \sigma\ \surd \longrightarrow P,\Phi \vdash \sigma'\ \surd$$

This theorem states an invariance of the conformance relation in a well-typed program. The proof is by induction on all the frames and then by case distinction on the instruction set. The proof of this invariance for the *Invoke* instruction is complex since it is involved with the method lookup and the structure of frames.

Theorem 4.3 and Theorem 4.4 together say that all states during the execution of the program P conform to Φ if the Micro-Dalvik VM starts by executing *M* in a correct start state. Therefore, combined with Theorem4.2, we can conclude the proof of the type safety theorem 4.1.

5. Related Work

Stata *et al.* give a first paper formalization of Java bytecode verification using typing rules to describe bytecode verifier [5]. Freund *et al.* extend the work by Stata *et al.* to a substantial subset of the JVM [6]. Coglio *et al.* give the first machine-checked bytecode verifier in SPECWARE [10]. They take the perspective of a constraint solving for bytecode verification, a more general view than a dataflow analysis. However, they neither describe how to generate a constraint inequality based on their lattice structure, nor prove the monotonicity of flow functions. Qian proves the correctness of an abstract algorithm for turning his type checking rules into a data flow analyzer [8]. Klein *et al.* are the first to formulate the bytecode verification algorithm as a direct instance of an abstract data flow framework for a nontrivial subset of the JVM and they give checked proof for the correctness of their BCV implementation in Isabelle/HOL.

Payet *et al.* define an operational semantics of Android activities including Dalvik bytecode [10]. This pen-and-pencil definition is in the process of implementation using a symbolic executor. In their formalization the Dalvik bytecode is assumed to be verified. Jeon *et al.* give a symbolic executor written in *Ocaml* code that operates on Dalvik bytecode. They specify a special return register in their register mapping, but we do not know how this special register differs from others in their implementation.

They only assume a lookup function for the rule of the invoke instruction. They do not consider bytecode verification either and do not model exception.

There are some research on modelling Android platform and lifestyle to perform static analysis of Dalvik bytecode. Arzt *et al.* give a novel and highly precise static taint-analysis system [13]. The work by Gibler *et al.* automatically finds potential leaksof sensitive information in Android applications on a massive scale [14]. Wognsen *et al.* give the formal verification for Dalvik bytecode and develops a prototype that translates Android applications to constraints in Python clauses [15]. Their work differs in the precision, efficiency, and scope of the tool that they implement to detect vulnerabilities in Android applications.

6. Conclusion

In this paper we give a provably-correct Micro-Dalvik bytecode verifier and implement it in the theorem proof assistant Isabelle/HOL. The work includes the operational semantics for a substantial set of Dalvik, construction of the semilattice structure and transfer functions for its type system. The bytecode verifier is realized as a result of concreting a general version of Kildall's algorithm with the semilattice and transfer functions. The correctness of the verifier is proved with respect to the type system which is proved to be type-safe using conforming relations. In the future we will cover threads and arrays.

Acknowledgments

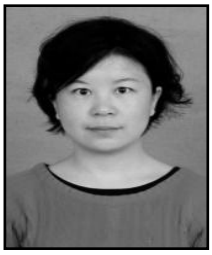
This work is supported by the National Natural Science Foundation of China (General Program) under Grant No. 61373039.

References

[1] T. Lindholm, F. Yellin, G. Bracha and A. Buckley, "The Java Virtual Machine Specification", Addison-Wesley, **(2013)**.
[2] Security Engineering Research Group, "Analysis of Dalvik Virtual machine and Class Path Library", Technical Report, Pakistan: Institute of Management Sciences Peshawar, **(2009)**.
[3] G. Klein and T. Nipkow, "A machine-checked model for a Java-like language, virtual machine, and compiler", ACM Transaction of Program Language and System, vol. 28, no. 4, **(2006)**, pp. 619-695.
[4] C. Pusch, "Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL", Tools and Algorithms for the Construction and Analysis of Systems Conference, Amsterdam, The Netherlands, **(1999)** March 22-28.
[5] R. Stata and M. Abadi, "A type system for Java bytecode subroutines", ACM Transaction of Program Language and System, vol. 21, no. 1, **(1999)**, pp. 90-137.
[6] S. N. Freund and J. C. Mitchell, "A formal framework for the Java bytecode language and verifier", Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Denver, CO, **(1999)** November 1-5.
[7] A. Coglio, A. Goldberg and Z. Y. Qian, "Toward a provably-correct implementation of the JVM bytecode verifier", Proceedings of the DARPA Information Survivability Conference and Exposition, Hilton Head, South Carolina, **(2000)** January 25-27.
[8] Z. Y. Qian, "Standard fixpoint iteration for Java bytecode verification", ACM Transaction of Program Language and System, vol. 22, no. 4, **(2000)**, pp. 638-672.
[9] A. Goldberg, "A specification of Java loading and bytecode verification", Proceedings of the 5th ACM conference on Computer and communications security, San Francisco, CA, (**1998**) November 3-5.
[10] E. Payet and F. Spoto, "An operational semantics of Android Activities", Proceedings Of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, San Deigo, California, **(2014)** January 20-21.
[11] J. Jeon, K. K. Micinski and J. S. Foster, "SymDroid: Symbolic Execution for Dalvik Bytecode", Technical Report, CS-TR-5022, Department of Computer Science, University of Maryland, College Park, **(2012)**.
[12] G. A. Kildall, "A unified approach to global program optimization", Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Boston, Massachusetts, **(1973)** October.

[13] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau and P. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps", vol. 49, no. 6, **(2014)**, pp. 259-269.
[14] C. Gibler, J. Crussell, J. Erickson and H. Chen. "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale", Proceedings of the 5th international conference on Trust and Trustworthy Computing, Vienna, Australia, **(2012)** June 13-15.
[15] E. R. Wognsen, H. S. Karlsen, M. C. Olesen and R. R. Hansen, "Formalization and Analysis of Dalvik Bytecode", Science of Computer Programming: special issue on Bytecode, Elsevier, vol. 92, **(2012)**, pp. 25-55.

Authors



Jiang Nan, received the MS degree from Hubei University of Technology in 2003. She worked as a visiting scholar in 2009 at Georgia Institute of Technology. She is currently a PhD candidate at Wuhan University and a full-time teacher in Hubei University of Technology. Her research interests include programming language formal methods, and trustworthy software.

He Yanxiang, received the BS and MS degrees from Wuhan University in 1975 and University of Oregon in 1986, respectively. He received the PhD degree from the Computer School of Wuhan University in 1999. He is currently a professor and a doctoral supervisor in Wuhan University. His research interests include trustworthy software, programming language, software engineering and distribution computing.

